Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №7

Технології паралельних обчислень

Тема: Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності

Виконав
студент групи ІП-11:
Панченко С. В.

Перевірила:
Стеценко І.В.

Київ 2024

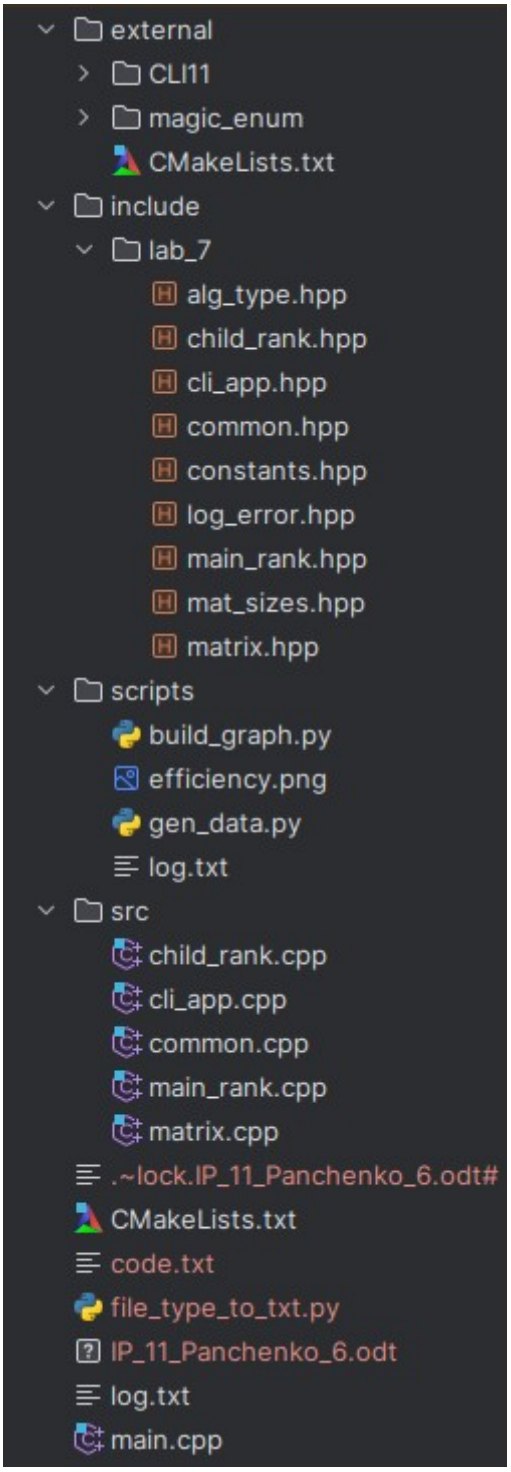
ЗМІСТ

1 ЗАВДАННЯ

1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. 40 балів.
3. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні методів обміну повідомленнями «один-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох». 60 балів.

2 ВИКОНАННЯ

2.1 Структура проєкту



External — папка для збереження зовнішніх бібліотек.

CLI11 — бібліотека дл легкої побудови command-line-interface’ів.

magic_enum — бібліотека для зручної роботи з enum`ами в C++.

alg_type.hpp — файл enum`а з типами алгоритмів.

child_rank.hpp child_rank.cpp — файли для фукнкцій, що відповідають за роботу дочірніх процесів.

cli_app.hpp, cli_app.cpp — файли для збереження класу консольного інтерфейсу.

common.hpp, common.cpp — файли для збереження функцій, що надають спільний функціонал для основного та дочірніх процесів.

constants.hpp — файл для глобальних констант.

log_error.hpp — файл макросів для логування та репортування помилок.

main_rank.hpp, main_rank.cpp — файли для функцій, що відровідають за роботу основного процесу.

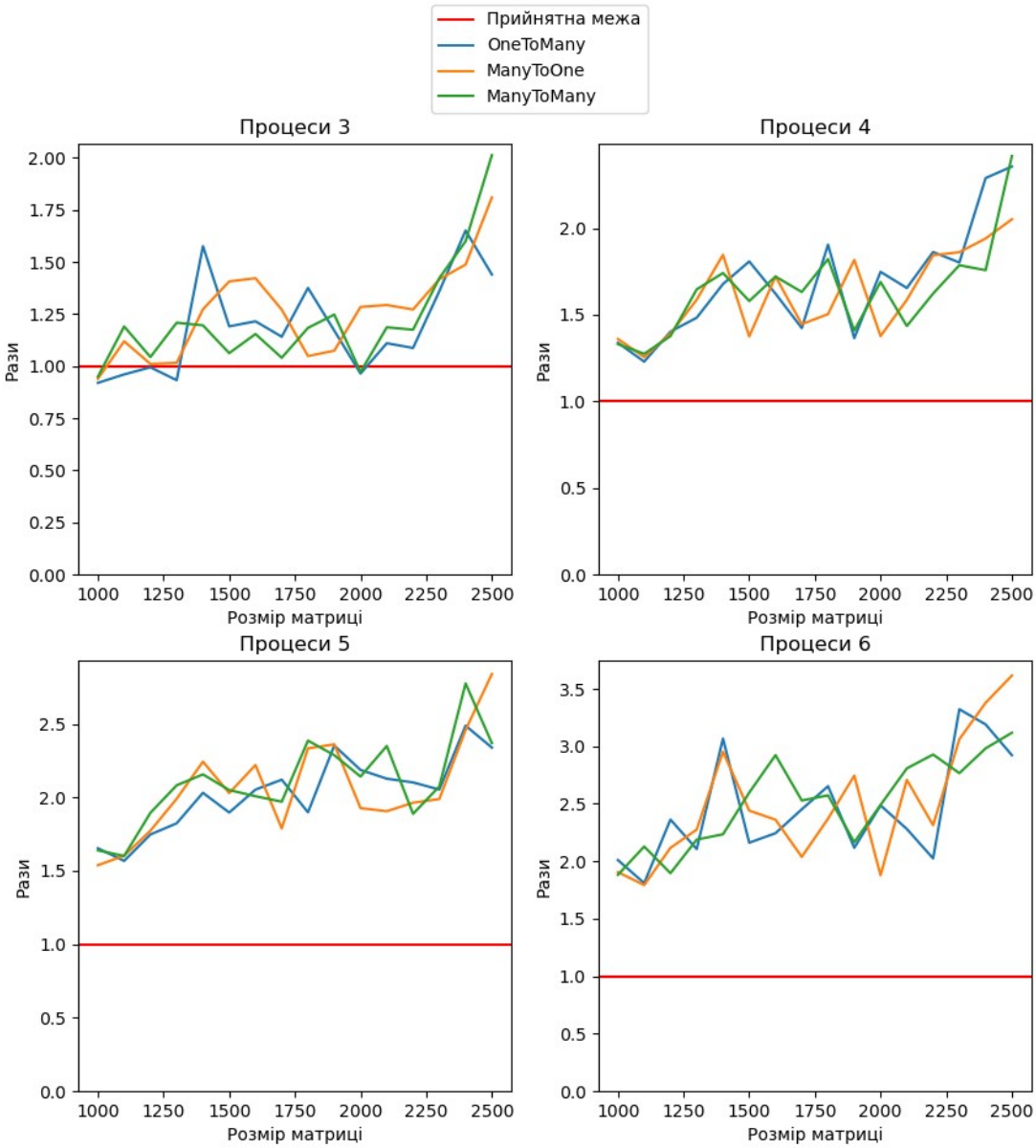
mat_sizes.hpp — файл для структури збереження розмірів матриць.

matrix.hpp, matrix.cpp — файл для класу матриць.

build_graph.py — файл для побудови графіків.

gen_data.py — файл для генерації результатів.

2.2 Результати



На графіках представлено результати тестування алгоритму множення матриць за допомогою різних методів передачі повідомлень. Тестування проведено для шести процесів, результати кожного з яких представлені окремо. Вісь абсцис (горизонтальна ось) показує розмір матриць, які множаться, а вісь ординат (вертикальна ось) відображає виміряний час виконання операції множення. Різні кольори ліній на графіках представляють різні методи передачі повідомлень:

- Червоний колір позначає "Прийнята межа" (threshold), що вказує на допустимий максимальний час виконання.

- Синій колір ("OneToOne") представляє метод передачі повідомлень від одного до одного.
- Зелений колір ("OneToMany") позначає метод передачі повідомлень від одного до багатьох.
- Помаранчевий колір ("ManyToOne") відображає метод передачі повідомлень від багатьох до одного.
- Синьо-зелений колір ("ManyToMany") представляє метод передачі повідомлень від багатьох до багатьох.

Загальні спостереження за графіками показують, що зі збільшенням розміру матриць час виконання операцій множення, як правило, зростає для всіх методів передачі повідомлень. Проте, ефективність кожного методу може відрізнятися в залежності від конкретного процесу та розміру матриць. Наприклад, у деяких випадках метод "ManyToMany" може показувати кращі результати, ніж інші методи, у інших — гірше. Тому з графіків не можна визначити, який метод краще.

3 ВИСНОВОК

Отже, під час лабораторної роботи опрацював схему комунікації між потоками з допомогою MPI. Побачили, що досягли суттєво прискорення, але особливої різниці між різними методами не спострігається.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-113 курсу

Панченка С. В

```
// include/lab_7/log_error.hpp

#ifndef ERROR_HPP
#define ERROR_HPP

#include <chrono>

#define LOG(msg, level) \
    do { \
        std::time_t now = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now()); \
        char timestamp[100]; \
        std::strftime(timestamp, sizeof(timestamp), "%Y/%m/%d %H:%M:%S", std::localtime(&now)); \
        std::cout <<"[" << timestamp << "] [" << __FILE__ << "] [" << __LINE__ << "] [" << level << "] [" << msg << "]" << std::endl; \
    } while(0)

#define LOG_INFO(msg) LOG(msg, "INFO")
#define LOG_WARN(msg) LOG(msg, "WARN")
#define LOG_ERROR(msg) LOG(msg, "ERROR")

#define ERROR(msg) \
    do {        \
        LOG_ERROR(msg);  \
        std::terminate();\
    } while(false)

#endif //ERROR_HPP



// include/lab_7/constants.hpp
```

```
#ifndef LAB6_CONSTANTS_H
#define LAB6_CONSTANTS_H

constexpr auto MAIN_PROC_RANK = 0;

// SYSTEM DEPENDANT
static constexpr auto MINIMUM_ALLOC_SIZE = 224 * 2;

static const std::string MAIN_SHARED_MEMORY_NAME = "eeeeeeeeeeeeeiomopnohjubuybnijsdbgiuarsbifbaeuigbeslgnuiesgnieangloersiugn";

#endif //LAB6_CONSTANTS_H



// include/lab_7/matrix.hpp

#ifndef MATRIX_HPP
#define MATRIX_HPP

#include <span>
#include <vector>
#include <iostream>

namespace mmat {
    template<typename Container>
    struct RectContainer {
        std::vector<Container> Data;
        friend std::ostream& operator<<(std::ostream& out, const RectContainer& obj) {
            out << "{ ";
            for(const auto& row : obj.Data) {
```

```
                out << "{ ";
                for(const auto& el : row) {
                    out << el << " ";
                }
                out << "} ";
            }
            out << "}";
            return out;
        }
    };

    using RectSpan = RectContainer<std::span<double>>;
    using RectVector = RectContainer<std::vector<double>>;

    class MatrixSpan {
        public:
        MatrixSpan(const unsigned rows, const unsigned cols, double* ptr);

        public:
        unsigned get_rows() const;
        unsigned get_cols() const;
        std::span<double> get_at(const unsigned row) const;
        double get_at(const unsigned row, const unsigned col) const;
        RectVector mul(const MatrixSpan& other) const;
        void randomize();
        friend std::ostream& operator<<(std::ostream& out, const
MatrixSpan& mat);

        protected:
        std::span<double> data;
        unsigned rows = 0;
        unsigned cols = 0;
```

```
    };
}

#endif //MATRIX_HPP

// include/lab_7/cli_app.hpp

#ifndef CLI_HPP
#define CLI_HPP

#include <lab_7/alg_type.hpp>

#include <CLI/CLI.hpp>

class CliApp : public CLI::App {
    public:
    CliApp();

    public:
    AlgorithmType alg_type;
    unsigned size;
};

#endif //CLI_HPP


// include/lab_7/main_rank.hpp

#ifndef MAIN_RANK_HPP
#define MAIN_RANK_HPP

#include <chrono>
```

```
#include <boost/mpi.hpp>

namespace common {
	struct MatSizes;
}

enum class AlgorithmType;

namespace main_rank {
	struct AlgStatistic {
		int world_size = 0;
		std::chrono::system_clock::duration mpi_dur;
		std::chrono::system_clock::duration single_dur;
	};

	auto execute(
		const boost::mpi::communicator& world,
		const AlgorithmType& alg_type,
		const unsigned step_length,
		const common::MatSizes& sizes
	) -> AlgStatistic;
}

#endif //MAIN_RANK_HPP


// include/lab_7/mat_sizes.hpp

#ifndef MAT_SIZES_HPP
#define MAT_SIZES_HPP

namespace common {
```

```
        struct MatSizes {
                unsigned first_rows = 0;
                unsigned first_cols = 0;
                unsigned second_rows = 0;
                unsigned second_cols = 0;
        };
}

#endif //MAT_SIZES_HPP



// include/lab_7/child_rank.hpp

#ifndef CHILD_RANK_HPP
#define CHILD_RANK_HPP

#include <boost/mpi.hpp>

namespace common {
        struct MatSizes;
}

enum class AlgorithmType;

namespace child_rank {

        auto execute(
                const boost::mpi::communicator& world,
                const AlgorithmType& alg_type,
                const unsigned step_length,
                const common::MatSizes& sizes
        ) -> void;
```

```
}

#endif //CHILD_RANK_HPP


// include/lab_7/common.hpp

#ifndef COMMON_HPP
#define COMMON_HPP

#include <lab_7/matrix.hpp>

#include <boost/mpi.hpp>
#include <boost/interprocess/managed_shared_memory.hpp>

namespace inter = boost::interprocess;

namespace common {

    struct MatSizes;

    auto get_steps(
            const unsigned rank,
            const unsigned first_rows,
            const unsigned step_length
    ) -> unsigned;

    auto get_task_memory_name(
            const unsigned child_rank
    ) -> std::string;
```

```
        auto get_main_matrices(
                const inter::managed_shared_memory& main_memory,
                const inter::managed_shared_memory::handle_t handle,
                const MatSizes& sizes
        ) -> std::tuple<mmat::MatrixSpan, mmat::MatrixSpan>;

        auto calculate_partial_result(
                const boost::mpi::communicator& world,
                const unsigned step_length,
                const MatSizes& sizes,
                const mmat::MatrixSpan& first,
                const mmat::MatrixSpan& second
        ) -> inter::managed_shared_memory::handle_t;

}

#endif //COMMON_HPP


// include/lab_7/alg_type.hpp

#ifndef ALG_TYPE_HPP
#define ALG_TYPE_HPP

enum class AlgorithmType {
        OneToMany,
        ManyToOne,
        ManyToMany
};

#endif //ALG_TYPE_HPP
```

```
// src/cli_app.cpp

#include <lab_7/cli_app.hpp>
#include <lab_7/log_error.hpp>
#include <lab_7/main_rank.hpp>
#include <lab_7/child_rank.hpp>
#include <lab_7/mat_sizes.hpp>

#include <boost/mpi.hpp>
#include <magic_enum.hpp>

static const auto MAPPED_ALG_TYPES = [] {
	constexpr auto values = magic_enum::enum_values<AlgorithmType>();
	constexpr auto names = magic_enum::enum_names<AlgorithmType>();
	auto mapped = std::unordered_map<std::string, AlgorithmType>();
	for(auto i = 0u; i < values.size(); ++i) {
		mapped.insert_or_assign(names[i].data(), values[i]);
	}
	return mapped;
}();

auto main_logic(
	const common::MatSizes& sizes,
	const AlgorithmType& alg_type
) -> std::optional<main_rank::AlgStatistic> {

	if(sizes.first_cols != sizes.second_rows) {
		ERROR("Impossible to multiply");
	}

	const auto env = boost::mpi::environment();
	const auto world = boost::mpi::communicator();
```

```
        const auto subprocs_num = static_cast<unsigned>(world.size());
        if(subprocs_num <= 0) {
            ERROR("Number of tasks is less or equal zero");
        }

        const auto [tasks_num, step_length] = [&sizes, &subprocs_num] {
            const auto is_rows_less = sizes.first_rows < subprocs_num;
            const auto tasks_num = is_rows_less ? sizes.first_rows : subprocs_num;
            const auto step_length = is_rows_less ? 1 : tasks_num;
            return std::make_tuple(tasks_num, step_length);
        }();

        const auto rank = world.rank();
        const auto is_valid = rank <= tasks_num;
        const auto local = world.split(is_valid ? 0 : 1);

        if(not is_valid) {
            return std::nullopt;
        }

        if(rank == 0) {
            return main_rank::execute(local, alg_type, step_length, sizes);
        }
        child_rank::execute(local, alg_type, step_length, sizes);
        return std::nullopt;
    }

    CliApp::CliApp() : CLI::App("lab_6") {
        add_option("--size,-s", size, "Matrix size")
            ->required(true)
```

```
            ->check(CLI::PositiveNumber);
        add_option("--type,-t", alg_type, "Algorithm type")
            ->transform(CLI::CheckedTransformer(MAPPED_ALG_TYPES,
CLI::ignore_case))
            ->required(true);

        parse_complete_callback([this] {
            const auto res = main_logic({size, size, size, size}, alg_type);
            if(res) {
                const auto& statistic = res.value();
                const auto nano_mpi =
static_cast<double>(statistic.mpi_dur.count());
                const auto nano_single =
static_cast<double>(statistic.single_dur.count());
                const auto efficiency = nano_single / nano_mpi;
                auto log_file = std::ofstream("log.txt", std::ios::app);
                if(not log_file.is_open()) {
                    ERROR("Error opening log file");
                }
                log_file << std::format("{{"
                                        "\"procs_num\": {}, "
                                        "\"mat_size\": {}, "
                                        "\"alg_type\": \"{}\", "
                                        "\"mpi_nanos\": {}, "
                                        "\"single_nanos\": {}, "
                                        "\"efficiency\": {} "
                                        "}}", statistic.world_size, size,
                magic_enum::enum_name(alg_type), nano_mpi, nano_single,
efficiency)
                    << std::endl;
            }
        });
```

```
}


// src/matrix.cpp

#include <lab_7/matrix.hpp>
#include <random>

namespace mmat {
    MatrixSpan::MatrixSpan(const unsigned rows, const unsigned cols, double* ptr) {
        this->rows = rows;
        this->cols = cols;
        data = std::span(ptr, rows * cols);
    }

    unsigned MatrixSpan::get_rows() const {
        return rows;
    }

    unsigned MatrixSpan::get_cols() const {
        return cols;
    }

    std::span<double> MatrixSpan::get_at(const unsigned row) const {
        return data.subspan(row * cols, cols);
    }

    void MatrixSpan::randomize() {
        auto rd_ = std::random_device();;
        auto rng_ = std::mt19937(rd_());
        auto uni_ = std::uniform_real_distribution<double>(0, 1);
```

```
        for(auto& value : data) {
            value = uni_(rng_);
        }
    }

    std::ostream& operator<<(std::ostream& out, const MatrixSpan& mat) {
        out << "{ ";
        for(auto i = 0u; i < mat.rows; ++i) {
            const auto row = mat.get_at(i);
            out << "{ ";
            for(auto j = 0u; j < mat.cols; ++j) {
                out << row[j] << " ";
            }
            out << "} ";
        }
        out << "}";
        return out;
    }

    double MatrixSpan::get_at(const unsigned row, const unsigned col) const
{
        return data[row * cols + col];
    }

    RectVector MatrixSpan::mul(const MatrixSpan& other) const {
        auto result = std::vector<std::vector<double>>(rows);
        for(auto i = 0; i < rows; ++i) {
            auto row = std::vector<double>(other.cols);
            for(auto j = 0; j < other.cols; ++j) {
                for(auto k = 0; k < cols; ++k) {
                    row[j] += this->get_at(i, k) * other.get_at(k, j);
                }
```

```
                }
                result[i] = std::move(row);
            }
            return {result};
        }
    }

    // src/main_rank.cpp

    #include <lab_7/main_rank.hpp>
    #include <lab_7/matrix.hpp>
    #include <lab_7/common.hpp>
    #include <lab_7/constants.hpp>
    #include <lab_7/log_error.hpp>
    #include <lab_7/alg_type.hpp>
    #include <lab_7/mat_sizes.hpp>

    #include <boost/interprocess/managed_shared_memory.hpp>

    namespace main_rank {

        struct MainMemoryGuard {
            MainMemoryGuard() {

inter::shared_memory_object::remove(MAIN_SHARED_MEMORY_NAME.data())
;
            }
            ~MainMemoryGuard() {

inter::shared_memory_object::remove(MAIN_SHARED_MEMORY_NAME.data())
```

```
;
        }
    };

    struct SharedMemoryRemover {
        SharedMemoryRemover()=default;
        SharedMemoryRemover(std::string&& _name) :
name{std::move(_name)} {}
        ~SharedMemoryRemover() {
            inter::shared_memory_object::remove(name.data());
        }
        protected:
        std::string name;
    };

    static auto multiply_single(
        const mmat::MatrixSpan& first,
        const mmat::MatrixSpan& second
    ) -> std::tuple<
            std::chrono::system_clock::duration,
            mmat::RectVector
        > {
        const auto start_time = std::chrono::system_clock::now();
        auto result = first.mul(second);
        const auto end_time = std::chrono::system_clock::now();
        const auto duration = end_time - start_time;
        return std::make_tuple(duration, std::move(result));
    }

    static auto create_matrices(
        const common::MatSizes& sizes
    ) -> std::tuple<
```

```
            inter::managed_shared_memory,
            inter::managed_shared_memory::handle_t,
            mmat::MatrixSpan,
            mmat::MatrixSpan
        > {
            const auto first_mat_space = sizeof(double) * sizes.first_rows * sizes.first_cols;
            const auto second_mat_space = sizeof(double) * sizes.second_rows * sizes.second_cols;
            const auto alloc_size = first_mat_space + second_mat_space;
            auto shared_memory = inter::managed_shared_memory(inter::create_only, MAIN_SHARED_MEMORY_NAME.data(),
                                alloc_size + MINIMUM_ALLOC_SIZE, nullptr);
            if(const auto free_size = shared_memory.get_free_memory(); alloc_size > free_size) {
                ERROR(std::format("Free size {} is less than alloc size {}", free_size, alloc_size));
            }
            const auto alloc_ptr = shared_memory.allocate(alloc_size);
            const auto main_handle = shared_memory.get_handle_from_address(alloc_ptr);
            auto [first_mat, second_mat] = get_main_matrices(shared_memory, main_handle, sizes);
            first_mat.randomize();
            second_mat.randomize();
            return std::make_tuple(std::move(shared_memory), main_handle, first_mat, second_mat);
        }

        static auto send_main_handle(
            const boost::mpi::communicator& world,
```

```
            const AlgorithmType& alg_type,
            const inter::managed_shared_memory::handle_t handle
        ) -> void {
            switch(alg_type) {
                case AlgorithmType::OneToMany: {
                    auto handle_copy = handle;
                    boost::mpi::broadcast(world, handle_copy, MAIN_PROC_RANK);
                    break;
                }
                case AlgorithmType::ManyToOne: {
                    for(auto task_rank = 1; task_rank < world.size(); ++task_rank) {
                        boost::mpi::gather(world, handle, task_rank);
                    }
                    break;
                }
                case AlgorithmType::ManyToMany: {
                    auto handles = std::vector<inter::managed_shared_memory::handle_t>();
                    boost::mpi::all_gather(world, handle, handles);
                    break;
                }
            }
        }

        static auto add_partial_result(
            const unsigned first_rows,
            const unsigned second_cols,
            const unsigned step_length,
            const unsigned task_rank,
            const inter::managed_shared_memory::handle_t task_handle,
```

```
            std::vector<inter::managed_shared_memory>& tasks_memories,
            std::vector<SharedMemoryRemover>& tasks_removers,
            std::vector<std::span<double>>& result
        ) -> void {

            const auto steps = common::get_steps(task_rank, first_rows, step_length);
            auto memory_name = common::get_task_memory_name(task_rank);
            auto child_memory = inter::managed_shared_memory(inter::open_only, memory_name.data());
            tasks_removers.emplace_back(std::move(memory_name));
            const auto ptr = child_memory.get_address_from_handle(task_handle);
            const auto partial_result = mmat::MatrixSpan(steps, second_cols, static_cast<double*>(ptr));
            const auto initial_index = task_rank;
            for(auto i = 0u; i < steps; ++i) {
                result[initial_index + i * step_length] = partial_result.get_at(i);
            }
            tasks_memories.emplace_back(std::move(child_memory));
        }

        static auto collect_results(
            const boost::mpi::communicator& world,
            const AlgorithmType& alg_type,
            const unsigned step_length,
            const unsigned first_rows,
            const unsigned second_cols,
            const inter::managed_shared_memory::handle_t main_task_handle
        ) -> std::tuple<
                std::vector<inter::managed_shared_memory>,
```

```
            std::vector<SharedMemoryRemover>,
            mmat::RectSpan
        > {
        const auto tasks_num = world.size();
        auto tasks_memories = std::vector<inter::managed_shared_memory>();
        auto tasks_removers = std::vector<SharedMemoryRemover>();
        tasks_memories.reserve(tasks_num);
        tasks_removers.reserve(tasks_num);
        auto result = std::vector<std::span<double>>(first_rows);

        add_partial_result(first_rows, second_cols, step_length, MAIN_PROC_RANK, main_task_handle, tasks_memories, tasks_removers, result);

        switch(alg_type) {
            case AlgorithmType::OneToMany: {
                for(auto task_rank = 1; task_rank < tasks_num; ++task_rank) {
                    auto child_handle = inter::managed_shared_memory::handle_t();
                    boost::mpi::broadcast(world, child_handle, task_rank);
                    add_partial_result(first_rows, second_cols, step_length, task_rank, child_handle, tasks_memories, tasks_removers, result);
                }
                break;
            }
            case AlgorithmType::ManyToOne: {
                auto handles = std::vector<inter::managed_shared_memory::handle_t>();
                boost::mpi::gather(world, main_task_handle, handles, MAIN_PROC_RANK);
```

```
                        for(auto task_rank = 1; task_rank < tasks_num; ++task_rank) {
                            boost::mpi::gather(world, main_task_handle, task_rank);
                            add_partial_result(first_rows, second_cols, step_length, task_rank, handles[task_rank], tasks_memories, tasks_removers, result);
                        }
                        break;
                    }
                    case AlgorithmType::ManyToMany: {
                        auto handles = std::vector<inter::managed_shared_memory::handle_t>();
                        boost::mpi::all_gather(world, main_task_handle, handles);
                        for(auto task_rank = 1; task_rank < tasks_num; ++task_rank) {
                            add_partial_result(first_rows, second_cols, step_length, task_rank, handles[task_rank], tasks_memories, tasks_removers, result);
                        }
                        break;
                    }
                }
                return std::make_tuple(std::move(tasks_memories), std::move(tasks_removers), mmat::RectSpan{std::move(result)});
            }

            static auto check_results(
                const mmat::RectVector& single_res,
                const mmat::RectSpan& mpi_res
            ) -> void {
                if(single_res.Data.size() != mpi_res.Data.size()) {
                    ERROR("Rows size does not match");
```

```
                }
                for(auto row = 0u; row < single_res.Data.size(); ++row) {
                        const auto& row_single = single_res.Data[row];
                        const auto& row_mpi = mpi_res.Data[row];
                        if(row_single.size() != row_mpi.size()) {
                                ERROR("Cols size does not match");
                        }
                        for(auto col = 0u; col < row_single.size(); ++col) {
                                if(std::abs(row_single[col] - row_mpi[col]) > std::numeric_limits<double>::epsilon()) {
                                        ERROR("Elements does not match");
                                }
                        }
                }
        }

        auto execute(
                const boost::mpi::communicator& world,
                const AlgorithmType& alg_type,
                const unsigned step_length,
                const common::MatSizes& sizes
        ) -> AlgStatistic {

                const auto guard = MainMemoryGuard();

                const auto [main_memory, main_handle, first_mat, second_mat] = create_matrices(sizes);
                const auto [single_duration, single_result] = multiply_single(first_mat, second_mat);

                const auto mpi_start_time = std::chrono::system_clock::now();
```

```
                send_main_handle(world, alg_type, main_handle);
                const auto task_handle = calculate_partial_result(world, step_length, sizes, first_mat, second_mat);
                const auto [child_memories, child_memory_removers, mpi_result] = collect_results(
                        world, alg_type, step_length, sizes.first_rows, sizes.second_cols, task_handle);

                const auto mpi_dur = std::chrono::system_clock::now() - mpi_start_time;
                check_results(single_result, mpi_result);
                return AlgStatistic{world.size(), mpi_dur, single_duration};
            }

        }

        // src/common.cpp

        #include <lab_7/common.hpp>
        #include <lab_7/constants.hpp>
        #include <lab_7/mat_sizes.hpp>

        namespace common {

            // This is just random string, not a special token
            static const std::string TASK_SHARED_MEMORY_NAME = "fdogsdniubu43769223ndsosdfijspofmakanapmk_";

            auto get_steps(
                const unsigned rank,
                const unsigned first_rows,
                const unsigned step_length
```

```
        ) -> unsigned {
            const auto len = static_cast<double>(first_rows - rank);
            return static_cast<unsigned>(std::ceil(len / step_length));
        }

        auto get_task_memory_name(
            const unsigned child_rank
        ) -> std::string {
            return TASK_SHARED_MEMORY_NAME + std::to_string(child_rank);
        }

        auto get_main_matrices(
            const inter::managed_shared_memory& main_memory,
            const inter::managed_shared_memory::handle_t handle,
            const MatSizes& sizes
        ) -> std::tuple<mmat::MatrixSpan, mmat::MatrixSpan> {

            const auto first_mat_space = sizeof(double) * sizes.first_rows * sizes.first_cols;
            const auto ptr = static_cast<char*>(main_memory.get_address_from_handle(handle));
            auto first_mat = mmat::MatrixSpan(sizes.first_rows, sizes.first_cols, reinterpret_cast<double*>(ptr));
            auto second_mat = mmat::MatrixSpan(sizes.second_rows, sizes.second_cols, reinterpret_cast<double*>(ptr + first_mat_space));
            return std::make_tuple(first_mat, second_mat);
        }

        auto calculate_partial_result(
            const boost::mpi::communicator& world,
            const unsigned step_length,
```

```
            const MatSizes& sizes,
            const mmat::MatrixSpan& first,
            const mmat::MatrixSpan& second
        ) -> inter::managed_shared_memory::handle_t {

            const auto rank = static_cast<unsigned>(world.rank());
            const auto steps = get_steps(rank, sizes.first_rows, step_length);
            const auto alloc_memory = sizeof(double) * steps * sizes.second_cols;
            const auto memory_name = get_task_memory_name(rank);
            inter::shared_memory_object::remove(memory_name.data());
            auto child_memory = inter::managed_shared_memory(inter::create_only,
                memory_name.data(), alloc_memory + MINIMUM_ALLOC_SIZE, nullptr);
            const auto ptr = child_memory.allocate(alloc_memory);
            const auto child_handle = child_memory.get_handle_from_address(ptr);
            const auto child_result = mmat::MatrixSpan(steps, sizes.second_cols, static_cast<double*>(ptr));
            for(auto row_index = rank, i = 0u;
                i < steps; row_index += step_length, ++i) {
                auto row = child_result.get_at(i);
                for(auto j = 0; j < sizes.second_cols; ++j) {
                    auto value = 0.0;
                    for(auto k = 0; k < sizes.first_cols; ++k) {
                        value += first.get_at(row_index, k) * second.get_at(k, j);
                    }
                    row[j] = value;
                }
            }
```

```
            return child_handle;
        }
    }

    // src/child_rank.cpp

    #include <lab_7/child_rank.hpp>
    #include <lab_7/common.hpp>
    #include <lab_7/alg_type.hpp>
    #include <lab_7/constants.hpp>
    #include <lab_7/log_error.hpp>

    namespace child_rank {

        static auto read_main_handle(
            const boost::mpi::communicator& world,
            const AlgorithmType& alg_type
        ) -> inter::managed_shared_memory::handle_t {
            auto main_handle = inter::managed_shared_memory::handle_t();
            switch(alg_type) {
                case AlgorithmType::OneToMany: {
                    boost::mpi::broadcast(world, main_handle,
MAIN_PROC_RANK);
                    break;
                }
                case AlgorithmType::ManyToOne: {
                    for(auto task_rank = 1; task_rank < world.size(); +
+task_rank) {
                        if(task_rank == world.rank()) {
                            auto handles =
std::vector<inter::managed_shared_memory::handle_t>();
                            boost::mpi::gather(world, main_handle,
```

```
handles, task_rank);
                                main_handle =
handles[MAIN_PROC_RANK];
                            } else {
                                boost::mpi::gather(world, main_handle,
task_rank);
                            }
                        }
                        break;
                    }
                    case AlgorithmType::ManyToMany: {
                        auto handles =
std::vector<inter::managed_shared_memory::handle_t>();
                        boost::mpi::all_gather(world, main_handle, handles);
                        main_handle = handles[MAIN_PROC_RANK];
                        break;
                    }
                }
                return main_handle;
            }

            static auto send_child_handle(
                const boost::mpi::communicator& world,
                const AlgorithmType& alg_type,
                const inter::managed_shared_memory::handle_t handle
            ) -> void {
                switch(alg_type) {
                    case AlgorithmType::OneToMany: {
                        for(auto task_rank = 1; task_rank < world.size(); +
+task_rank) {
                            auto handle_copy = handle;
                            boost::mpi::broadcast(world, handle_copy,
```

```
task_rank);
                        }
                        break;
                    }
                    case AlgorithmType::ManyToOne: {
                        for(auto task_rank = 0; task_rank < world.size(); ++task_rank) {
                            if(task_rank == world.rank()) {
                                auto payloads = std::vector<inter::managed_shared_memory::handle_t>();
                                boost::mpi::gather(world, handle, payloads, task_rank);
                            } else {
                                boost::mpi::gather(world, handle, task_rank);
                            }
                        }
                        break;
                    }
                    case AlgorithmType::ManyToMany: {
                        auto handles = std::vector<inter::managed_shared_memory::handle_t>();
                        boost::mpi::all_gather(world, handle, handles);
                        break;
                    }
                }
            }

            auto execute(
                const boost::mpi::communicator& world,
                const AlgorithmType& alg_type,
                const unsigned step_length,
                const common::MatSizes& sizes
```

```
        ) -> void {
            const auto main_handle = read_main_handle(world, alg_type);
            const auto main_shared_memory = inter::managed_shared_memory(inter::open_only,
MAIN_SHARED_MEMORY_NAME.data());
            const auto [first, second] = get_main_matrices(main_shared_memory, main_handle, sizes);
            const auto child_handle = calculate_partial_result(world, step_length, sizes, first, second);
            send_child_handle(world, alg_type, child_handle);
        }
    }


    // scripts/gen_data.py

    import os

    processes = [3, 4, 5, 6]
    sizes = [i for i in range(1000, 2600, 100)]
    types_block = ['OneToMany', 'ManyToOne', 'ManyToMany']
    for _ in range(5):
        for t in processes:
            for s in sizes:
                for tb in types_block:
                    os.system(f'mpiexec -np {t} ../cmake-build-release/lab_7 -s {s} -t {tb}')


    // scripts/build_graph.py

    import pandas as pd
```

```
import ast
import os
import matplotlib.pyplot as plt


def main() -> None:
    with open('log.txt', 'r') as file:
        data = list(map(lambda line: ast.literal_eval(line), file))

    pd.read_csv('log.txt', sep='\t')
    data = pd.DataFrame(data)

        one_to_many = data[data.alg_type == 'OneToMany'][['mat_size',
'procs_num', 'efficiency']].groupby(
        ['mat_size', 'procs_num']).mean()
        many_to_one = data[data.alg_type == 'ManyToOne'][['mat_size',
'procs_num', 'efficiency']].groupby(
        ['mat_size', 'procs_num']).mean()
         many_to_many = data[data.alg_type == 'ManyToMany'][['mat_size',
'procs_num', 'efficiency']].groupby(
        ['mat_size', 'procs_num']).mean()

    rows = 2
    cols = 2

    figure, axis = plt.subplots(rows, cols, figsize=(10, 10))
    for index in range(0, rows * cols):
        threads_num = index + 3
        row = int(index / cols)
        col = index % cols
        ax = axis[row, col]
        ax.axhline(y=1, color='r', linestyle='-', label='Прийнятна межа')
```

```
                for (df, name) in [(one_to_many, 'OneToMany'), (many_to_one,
'ManyToOne'), (many_to_many, 'ManyToMany')]:
                    dd = df[df.index.get_level_values(1) == threads_num]
                    sizes = dd.index.get_level_values(0).values
                    ax.plot(sizes, dd.values, label=name)
                ax.set_title(f'Процеси {threads_num}')
                ax.set_xlabel('Розмір матриці')
                ax.set_ylabel('Рази')
                ax.set_ylim(ymin=0)
            handles, labels = axis[0, 0].get_legend_handles_labels()
            figure.legend(handles, labels, loc='upper center')
            #figure.tight_layout(rect=(0, 0, 1, 0.9))
            figure.savefig(f"efficiency.png")


        if __name__ == '__main__':
            main()
```